

The Ultimate Feature Flag Guide

What they are, how to use them and considerations for scaling

CloudBees.

Contents

Getting Started With Feature Flags

As any developer can tell you, deploying any code carries technical risk. Software might crash or bugs might emerge. Deploying features carries additional user-related risk. Users might hate the new features or run into account management issues. With traditional deployments, all of this risk is absorbed at once.

Thankfully, the leaders in software development (e.g., Facebook, Twitter, and Uber) pioneered a new method of reducing risk while still enabling rapid innovation and updates in the form of something called a **feature flag**. Since these companies made the practice mainstream, feature flag management has become an increasingly popular practice across companies today.

What is a Feature Flag?

Feature flags give developers the ability to separate these types of risks, handling them one at a time. They can deploy the new code into production, see how that goes, and then turn on the features later once it's clear the code is working as expected.

Simply put, a feature flag is a way to change a piece of software's functionality without changing and re-deploying its code. Feature flags involve creating a powerful "if statement" surrounding some chunk of functionality in software (pockets of source code).

The History of Feature Flags

Leading Web 2.0 companies with platforms and services that must maintain performance among high traffic levels led the way in developing and popularizing new deployment techniques. Facebook, in particular, is known as a pioneer of feature flags and for releasing massive amounts of code at scale.

While building its massive social network more than a decade ago, the company realized that its uptime and scale requirements could not be met with traditional site maintenance approaches. (A message saying the site was down while they deployed version 3.0 was not going to cut it).

Instead, Facebook just quietly rolled out a never-ending stream of updates without fanfare. Day to day, the site changed in subtle ways, adding and refining functionality.

At the time, this was a mean feat of engineering. Other tech titans such as Uber and Netflix developed similar deployment capabilities as well. The feature flag was philosophically fundamental to this development and set the standard for modern deployment maturity used by leading organizations everywhere today.

What Does a Feature Flag Look Like?

The following is a simple feature flag in code used on an e-commerce site. This software displays a “Happy holidays!” greeting message with a “see more holiday deals” call to action when it is instructed to. The developer has a way to toggle this functionality on and off without re-deploying the code for every holiday or across every instance where it should appear.

It would have a construct like this for the top of a page:

```
if(feature.isActive("holiday-greeting")) {
   print("Happy holidays," + user.name + "!");

}
```

And then perhaps something like this at the bottom:

```
if(feature.isActive("holiday-greeting")) {
   printLink("Click to see more holiday deals",
   "~/holidayDeals");

}
```

Anatomy of a Feature Flag

The following terms are helpful to review when discussing feature flags, also sometimes referred to as feature toggles, feature flippers, feature controls, or rollout flags.

Toggle Point

In this example, the toggle point is each occurrence of a "check for feature." `isActive("holiday-greeting")` represents a single toggle point. Since a feature is rarely just a few linear lines of code, turning a feature on and off can require many, many toggle points.

Toggle Router

The `feature.isActive` function takes the name of the feature flag as an argument and maps the toggle points to the state of that feature flag. In this example, it would be `feature.isActive("holiday-greeting")`. It acts as a coherent, single point of knowledge for the state of the feature across many toggle points.

Toggle Context

The toggle context represents contextual information that the router takes into account when computing the feature's state. In the "`holiday-greeting`" feature example above, the date would be the toggle context. Other examples might include the logged in user, the geolocation information of the user, a referring URL, etc.

Toggle Configuration

In addition to ambient (known) context, the results of the toggle router for a feature can also be controlled by a simple configuration, as seen in the above example — a manual turn off capability with a toggle configuration.

Ways to Use Feature Flags

Canary Launches

Alluding to the "canary in the coal mine," canary deployments occur when software developers release a new version of software to just a subset of users or systems. By enabling new software within only one part of the user base, developers can monitor any problems that arise without causing major disruption. Also known as "progressive delivery," this lets organizations keep general customer trust high while freeing developers to focus on innovating and delivering excellent new functionality to customers.

Testing in Production

Conventional wisdom has historically been that testing should be performed prior to production. However, internal test environments can never fully recreate production. Facebook, Netflix, and the other Web 2.0 organizations previously mentioned all understood that given their scale, they could not possibly recreate their production environments for testing purposes. As a result, they run QA in production. By using feature flags, their developers de-risk deploying functionality whose production behavior is unknown to them, which results in faster releases.

Turning off Functionality With a Kill Switch

In the same way that organizations can increase the number of users who see a feature, they can also decrease that number to zero, creating a kill switch. This capability comes in handy when a feature doesn't behave as planned. Developers can roll back the change and let life go back to normal for users. It's also useful for sunsetting and then decommissioning legacy features. Kill switches absorb much of the risk, stress, and overhead from a release for developers as it's simple to roll back a feature.

Running Experiments

Production experiments are another popular way of using feature flags. The most common example is an A/B test, through which an organization deploys two different versions of a feature to see which performs best. In the example of an e-commerce site, a brand might test whether a green or red "buy" button results in more transactions. The brand would deploy both button colors — using a feature flag to split its user base in half — and see which color performs the best. The possibilities for production experiments are endless and the results can significantly improve your application. When used correctly, experiments provide product owners and developers with a much shorter feedback loop with their end users, resulting in improved customer acquisition and retention.

Removing Risk From Migrations

Feature flags provide a low-risk alternative to the forklift upgrade of years past that often ended in fire drills. Rather than rewriting large swaths of code for a one-time migration that causes issues, developers can use feature flags to start building calls to the new service or database directly into the existing application. Then they can discreetly test out the new connection at a more secure time to see if it's working. When it is time to completely migrate, the code will have been in production for a long time and flipping the switch will be a non-event.

Improving Developer Productivity

Feature branching has been a long-time continuous integration practice that allows teams to work with their own copies of the code and, in general, merge all changes less frequently. However, long-lived feature branches can cause their own headaches as changes to the trunk and additions from other branches can result in what developers call "merge hell." It's easier for developers to add their branches back to the trunk — wrapped within a feature flag and toggled off — to create the best of both worlds for feature branching without the merge hell commonly associated with it.

Also, as flags allow independent development and deployment, developers no longer have to be on call to deploy features once they're finished. This frees up more time for development and improves developer satisfaction.

Feature Flags and Technical Debt

One of the historical knocks on the use of feature flags is that they create technical debt. This is an understandable sentiment since, if implemented by hand, they can lead to massive, ad hoc tangles of conditional logic in the codebase. And that can be downright nasty.

Technical debt is one of the various reasons why it can make sense to consider adopting third-party management systems. These systems can actually save you from technical debt, even as your own, homegrown solution may cause it. But even with such a system, you have to be careful.

In general, codebases tend toward entropy — they tend to rot unless you actively curate them. It's no different with your implementation of feature flag management. Do your best to group toggle points as close together as possible, rather than having features sprawl all over the application. Adhering to the SOLID principles will help with this, as will keeping your code (and feature flag logic) DRY. And make sure to ruthlessly cull outdated toggle points and routers and to plan to retire your feature flags.

How to Get Started

It may seem like turning on or off one little thing in an application is over-engineering. Consider the idea of application logging, however. If a developer were brand new to programming or writing some kind of toy implementation, it might actually be easier to just use their language's file API to dump some random text to a file than it would be to install and configure a full-blown logging framework.

But how long would that last? Would they hold out until they had to consider log levels? Different styles of appender? Multi-threading? Sooner or later, it would become more painful to keep rolling it themselves than going back and using an established solution.

The same goes for feature flag management. If a developer or team is new to it and dabbling, then it makes sense to implement it manually. They will develop an understanding of how it works and make better decisions later. But if they find themselves in over their heads, remember that mature, third-party feature flag management systems exist for a reason.

Feature Flags at Scale

Many organizations start using feature flags with homegrown systems that their developers create. These can be effective for smaller teams and companies. However, once a company is ready to scale their feature flags across their organization and make them a staple of their continuous delivery efforts, they need to consider whether their internal system will scale.

Asking the following questions can help determine which approach is the best fit:

Overhead

How many hours a week am I willing to devote to maintaining my internal flagging system? Am I comfortable dedicating developer hours to system maintenance and updates?

Visibility and Governance

How will technical and non-technical users be able to see which features are flagged, and when? How will I be able to ensure only correct users can flag items at the correct time, and those flags are retired appropriately?

Security and Data Privacy

How will I ensure that only the right users have access to flags? How do I create clear audit trails for my flags? How will I handle personally identifiable information (PII) within my flagging system analytics? Conversely, would I be comfortable storing my customers' PII on vendors' external platforms?

Analytics

How will I ensure I'm accurately getting feedback on my features in real time? Do I want to maintain another analytics dashboard, or do I want to integrate feature information into my current dashboards?

Developer Productivity

Are my developers most productive using an internal system as-is? Or would they benefit from using an external platform with reduced maintenance and likely greater developer-focused features? Is my current system as developer friendly as possible using new methodologies such as configuration as code?

As the number of feature flags used internally increases, the answers to these questions become increasingly complex. In fact, in a recent study, 90 percent of developers surveyed said they would consider moving to an external system as the overhead and complexity grew with their flag usage.

CloudBees. Whitepaper

Feature Flags are the Future of Continuous Delivery

While there are many considerations for adding feature flags into a software delivery lifecycle, the benefits of flagging most, if not all, features are clear. As the methodology and platforms mature, feature flags will become an integrated part of the software development infrastructure, and there may be a time when every feature is released wrapped in a flag for maximum safety and velocity.

At the same time, organizations will move from "if" to "how" in regard to better automating and integrating flags into their strategy. There is a maturity scale that should be considered for getting started with flags — from developing an internal system to most likely choosing an external platform to manage flags and integrate with the greater SDLC.

If you are just getting started, it will be a journey, no question. But the benefits will be worth the effort.

The following resources are tech stack-specific guides to getting started with feature flags:



- A guide to Android feature flags
- A guide to JavaScript feature flags
- A guide to NodeJS feature flags

Learn More

www.cloudbees.com/products

CloudBees CI is built on top of Jenkins, an independent community project. Read more about Jenkins at: www.cloudbees.com/jenkins/about

1120v04

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com